(2)

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A249 802

1. AGENCY USI

3. REPORT TYPE AND DATES COVERED
Final: 16 Oct 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Validation Summary Report: Alsys/German MoD, NATO SWG APSE Compiler for Sun3/SunOS, Version S3C1.82-02, Sun3/SunOS under CAIS (Host) to Sun3/SunOS under CAIS (Host) to Sun3/SunOS(Target), 911016I1.11233

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 099

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Alsys/German MoD, NATO SWG APSE Compiler for Sun3/SunOS, Version S3C1.82-02, Ottobrunn Germany, Sun3/SunOS under CAIS (Host) to Sun3/SunOS under CAIS (Host) to Sun3/SunOS(Target), ACVC 1.11.

92-11784

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-10-16.

Compiler Name and Version:	**NATO SWG APSE Compiler for Sun3/SunOS Version S3C1.82-02**
Host Computer System:	**Sun3/60 / SunOS Version 4.0.3 under CAIS Version 5.5D**
Target Computer System:	**Sun3/60 / SunOS Version 4.0.3**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #911016I1.11233 is awarded to Alsys. This certificate expires on 01 June 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

for

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR 099
19 November 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 911016I1.11233
Alsys / German MoD
NATO SWG APSE Compiler for Sun3/SunOS
Version S3C1.82-02
Sun3/SunOS under CAIS Host
Sun3/SunOS Target

== based on TEMPLATE Version 91-05-08 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-10-16.

Compiler Name and Version: **NATO SWG APSE Compiler for Sun3/SunOS Version S3C1.82-02**

Host Computer System: **Sun3/60 / SunOS Version 4.0.3 under CAIS Version 5.5D**

Target Computer System: **Sun3/60 / SunOS Version 4.0.3**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #911016I1.11233 is awarded to Alsys. This certificate expires on 01 June 1993.

This report has been reviewed and is approved.

Michael Tonndorf

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys GmbH & CO. KG.

Certificate Awardee: Alsys / German MoD

Ada Validation Facility: IABG mbH, Germany

ACVC Version: 1.11

Ada Implementation:

NATO SWG on APSE Compiler for Sun3/SunOS Version S3C1.82-02

Host Computer System:
Sun3/SunOS Version 4.0.3 under CAIS Version 5.5D

Target Computer System: Sun3/SunOS Version 4.0.3

Declaration:

We, the undersigned, declare that we have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Customer Signature	28.11.91 **Date**

Alsys GmbH & Co. KG
7500 Karlsruhe 51 Am Rüppurrer Schloß 7
Tel. 07 21/88 30 25 Fax 07 21/88 75 64

Certificate Awardee Signature	28.11.91 **Date**

Alsys GmbH & Co. KG
7500 Karlsruhe 51 Am Rüppurrer Schloß 7
Tel. 07 21/88 30 25 Fax 07 21/88 75 64

DER BUNDESMINISTER DER VERTEIDIGUNG
141

Bonn, 11. Dezember 1991

Im Auftrag

Wilde

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C241130..Y (11 tests) (*)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

B22005A..C and B22005Y (4 tests), respectively, check that control the characters SOH, STX, ETX, and NUL are illegal when outside of character literals, string literals, and comments; for this implementation those characters have a special meaning to the underlying system such that the test file is altered before being passed to the compiler. (See section 2.3.)

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C41401A checks that CONSTRAINT_ERROR is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the

same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107C..D (2 tests), CE2107H, and CE2107L apply function NAME to temporary sequential, direct, and text files in an attempt to associate multiple internal files with the same external file; USE_ERROR is raised because temporary files have no name.

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

EE2401D uses instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file of such types.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available; this implementation buffers output. (See section 2.3.)

CE3202A expects that function NAME can be applied to the standard input and output files; in this implementation these files have no names, and USE_ERROR is raised. (See section 2.3.)

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 28 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

B22005A..C and B22005P (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests, respectively, check that control characters SOH, STX, ETX, and NUL are illegal outside of character literals, string literals, and comments. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in a way that defeats the test objectives--either the characters alone, or together with any text that follows them on the line, are not passed to the compiler. Hence, B22005B and B22005P compile without error, while the other tests have syntactic errors introduced by the loss of test text.

B25002A, B26005A, and B27005A were graded passed by Evaluation Modification as directed by the AVO. These tests check that control characters SOH, STX, ETX, and NUL are illegal within of character literals, string literals, and comments, respectively. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in the following way: these characters, and except in the case of NUL any text that follows them on the line, are not passed to the compiler. The tests were thus graded without regard for the lines that contained one of these four control characters.

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempts to read at lines 87 and 101, respectively.

CE3202A was graded inapplicable by Evaluation Modification as directed by the AVO. This test applies function NAME to the standard input file, which in this implementation has no name; USE_ERROR is raised but not handled, so the test is aborted. The AVO ruled that this behavior is acceptable pending any resolution of the issue by the ARG.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact in Germany for technical and sales information about this Ada implementation system, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

Testing of this Ada implementation was conducted at the AVF's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a)	Total Number of Applicable Tests	3824	
b)	Total Number of Withdrawn Tests	95	
c)	Processed Inapplicable Tests	92	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	159	
f)	Total Number of Inapplicable Tests	251	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

ACVC 1.11 was run at IABG's premises as follows: With the customer's macro parameter file the customised ACVC 1.11 was produced. Then CAIS version 5.5D as supplied by the customer was loaded and installed on the candidate SUN 3/60 computer. Next the basic CAIS node model and the candidate Ada implementation were installed. Then the full set of tests was processed using a test driver provided by the customer and reviewed by the validation team. Tests were processed using one input stream at a time. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Compilation was made using the following parameter settings:

```
SOURCE  => "'CURRENT_USER'DOT(SRC)"
LIBRARY => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
LIST    => "'CURRENT_USER'DOT(LIS)"
LOG     => "'CURRENT_USER'DOT(LOG)"
```

The parameters SOURCE and LIBRARY do not have a default value and need to be specified anyway.

The default of the parameters LIST and LOG means that no listing, resp. no log output is to be produced. The values used for validation are CAIS pathnames in order to obtain the corresponding output in the file nodes specified by the respective pathnames.

Linking was made with the following parameter settings:

```
UNIT       => ...  -- Main Program to be linked
LIBRARY    => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
EXECUTABLE => "'CURRENT_USER'DOT(EXE)"
DEBUG      => NO
LOG        => "'CURRENT_USER'DOT(LOG)"
```

The parameters UNIT, LIBRARY and EXECUTABLE do not have a default value and need to be specified anyway.

The default of the parameter LOG means that no log output is to be produced. The value used for validation is a CAIS pathname in order to obtain the corresponding output in the file node specified by the pathname.

The default value for the parameter DEBUG is not used, since ALSYS has provided only the runtime system which does not include debugger support.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Data Cartridge and archived at the AVF.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	2147483648
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	SUN3_SUNOS
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGCHLD)
$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR1)
$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR2)
$FIELD_LAST	512
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	0.0
$GREATER_THAN_DURATION_BASE_LAST	200_000.0
$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+256
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+256
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16#0.8#E+32
$HIGH_PRIORITY	15

$ILLEGAL_EXTERNAL_FILE_NAME1	/nodir/file1
$ILLEGAL_EXTERNAL_FILE_NAME2	/wrongdir/file2
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-0.0
$LESS_THAN_DURATION_BASE_FIRST	-200_000.0
$LINE_TERMINATOR	ASCII.LF
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	18
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2_147_483_648
$MIN_INT	-2147483648
$NAME	NO_SUCH_TYPE
$NAME_LIST	SUN3_SUNOS
$NAME_SPECIFICATION1	/var/users/vali/result11/chape/X2120A
$NAME_SPECIFICATION2	/var/users/vali/result11/chape/X2120B

$NAME_SPECIFICATION3	/var/users/vali/result11/chape/X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483648
$NEW_SYS_NAME	SUN3_SUNOS
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	1.0/50.0
$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units must all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed by exporting the file contents to a SunOS file and executing that file, or by using appropriate tools of the SWG APSE (CLI, Debugger), or by using CAIS operations.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.
Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.
Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the SYSTEAM Ada Compiler, use the `compile_host` command.

compile_host **Command Description**

Format

```
PROCEDURE compile_host (

    source               : string           ;
    analyze_dependency   : yes_no_answer    := no;
    check                : yes_no_answer    := yes;
    copy_source          : yes_no_answer    := yes;
    given_by             : source_choices   := pathname;
    inline               : yes_no_answer    := yes;
    library              : pathname_type
                                  := default_library;
    list                 : pathname_type    := nolist;
    log                  : pathname_type    := nolog;
    machine_code         : yes_no_answer    := no;
    optimize             : yes_no_answer    := yes;
```

`=> yes, ...)` command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.
If `copy_source => no` is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.
`copy_source => yes` cannot be specified together with `analyze_dependency`.

`given_by : source_choices := pathname`
`given_by => pathname` indicates that the string of the source parameter is to be interpreted as a pathname.
`given_by => unique_identifier` indicates that the string of the source parameter is to be interpreted as a unique identifier.
By default it is interpreted as a pathname.

`inline : yes_no_answer := yes`
Controls whether inline expansion is performed as requested by PRAGMA inline. If you specify no these pragmas are ignored.
By default, inline expansion is performed.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `compile_host` command needs write access to the library.
The default is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`list : pathname_type := nolist`
Controls whether a listing is written to the given file.
By default, the compile command does not produce a listing file.

`log : pathname_type := nolog`
Controls whether the Compiler appends additional messages onto the specified file.
By default, no additional messages are written.

`machine_code : yes_no_answer := no`
Controls whether machine code is appended at the listing file. `machine_code` has no effect if `list` is `nolist` or `analyze_dependency => yes` is specified.
By default, no machine code is appended at the listing file.

`optimize : yes_no_answer := yes`
Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.
By default, full optimization is done.

complete_host Command Description

Format

```
PROCEDURE complete_host (

     unit              : unitname_type        ;
     check             : yes_no_answer        := yes;
     inline            : yes_no_answer        := yes;
     library           : pathname_type
                                   := default_library;
     list              : pathname_type        := nolist;
     log               : pathname_type        := nolog;
     machine_code      : yes_no_answer        := no;
     optimize          : yes_no_answer        := yes);
```

Description

The `complete_host` command invokes the SYSTEAM Ada Completer. The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

By default, the Completer is invoked implicitly by the `link_host` command. In normal cases there is no need to invoke it explicitly.

Parameters

`unit : unitname_type`
specifies the unit whose execution closure is to be completed.

`check : yes_no_answer := yes`
Controls whether all run-time checks are suppressed. If you specify no this is equivalent to the use of PRAGMA suppress for all kinds of checks.
By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

`inline : yes_no_answer := yes`
Controls whether inline expansion is performed as requested by PRAGMA inline. If no is specified, these pragmas are ignored. By default, inline expansion is performed.

`library : pathname_type := default_library`

The set of units to be checked for recompilation or new compilation is described by specifying one or more units and the kind of a closure which is to be built on them. In many cases you will simply specify your main program.

The automatic recompilation of obsolete units is supported by the `recompile_host` command. It determines the set of obsolete units and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI.

The recompilation is performed using the copy of the obsolete units which is (by default) stored in the library. (If the user does not want to hold a copy of the sources the `recompile_host` command offers the facility to use the original source.)

The automatic compilation of modified sources is supported by the `autocompile_host` command. It determines the set of modified sources and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI. The basis of both the `recompile_host` and the `autocompile_host` command is the information in the library about the dependencies of the concerned units. Thus neither of these commands can handle the compilation of units which have not yet been entered in the library.

The automatic compilation of new sources is supported by the `compile_host` command together with the analyze_dependency parameter. This command is able to accept a set of sources in any order. It makes a syntactical analysis of the sources and determines the dependencies. The units "compiled" with this command are entered into the library, but only their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the `autocompile_host` command. They *cannot* be recompiled using the `recompile_host` command because the `recompile_host` command only recompiles units which were already compiled.

The next sections explain the usage of the `recompile_host` command, the `autocompile_host` command, and the `compile_host` command with `analyze_dependency => yes`.

4.3.1 Recompiling Obsolete Units

The `recompile_host` command supports the automatic recompilation of units which became obsolete because of the (re)compilation of units they depend on. The command gets as a parameter a set of units which are to be used to form the closure of units to be recompiled. The kind of the closure can be specified. The `recompile_host` command generates a command file with a sequence of compile commands to recompile the

In the command file each recompilation of a unit is executed under the condition that the recompilation of other units it depends on was successful. Thus useless recompilations are avoided. The generated command file only works correctly if the library was not modified since the command file was generated.

Note: If a unit from a parent library is obsolete it is compiled in the sublibrary in which the `recompile_host` command is used. In this case a later recompilation in the parent library may be hidden afterwards.

Parameters

`unit : unitname_type`
Specifies the unit whose closure is to be built.

`output : pathname_type`
Specifies the name of the generated command file.

`body_ind : yes_no_answer := no`
specifies that `unit` stands for the secondary unit with that name. By default, `unit` denotes the library unit. If `unit` specifies a subunit, the `body_ind` parameter need not be specified.

`bodies_only : yes_no_answer := no`
Controls whether all units of the closure are recompiled (default) or only the secondary units. This parameter is only effective if `conditional => no` is specified.

`check : yes_no_same_answer := same`
`check => same` means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the `compile_host` command. Otherwise the given value for the check parameter is included in the command file. By default the parameter value of the last compilation is included.

`closure : closure_choices := execute`
Controls the kind of the closure which is built and which is the basis for the investigation for obsolete units. `closure => noclosure` means that only the specified unit is checked. `closure => compile` means that only those units on which the specified unit transitively depends are regarded. `closure => execute` means that - in addition - all related secondary units and the units they depend on are regarded. If `closure => tree` is specified, a warning is issued stating that this is not meaningful for this command and that the default value is taken instead.
By default, the execution closure is built.

4.3.2 Compiling New Sources

The autocompile_host command supports the automatic compilation of units for which a new source exists. The command receives as parameter a unit which is to be used to form the closure of units to be processed. The kind of closure can be specified. For every unit in the closure, the autocompile_host checks whether there exists a newer source than that which was used for the last compilation. It generates a command file with a sequence of compile_host commands to compile the units for which a newer source exists. If a unit to be compiled depends on another unit which is obsolete or which will become obsolete and for which no newer source exists, the autocompile_host command always adds an appropriate compile_host (...., recompile => yes, ...) command to make it current; the recompile parameter controls which other obsolete units are recompiled, and can indeed be used to specify that the same recompilations are done as if the recompile_host command was applied subsequently. The generated command file is in fact an Ada program using the facilities of the package CLI_INTERFACE provided by the SWG APSE CLI. The name of the command file can be specified using the output parameter.

autocompile_host Command Description

Format

```
PROCEDURE autocompile_host (

        unit               : unitname_type          ;
        output             : pathname_type          ;
        body_ind           : yes_no_answer       := no;
        bodies_only        : yes_no_answer       := no;
        check              : yes_no_same_answer  := same;
        closure            : closure_choices     := execute;
        conditional        : yes_no_answer       := yes;
        copy_source        : yes_no_answer       := yes;
        inline             : yes_no_same_answer  := same;
        library            : pathname_type
                                       := default_library;
        list               : pathname_type       := nolist;
        log                : pathname_type       := nolog;
        machine_code       : yes_no_answer       := no;
        optimize           : yes_no_same_answer  := same;
        recompile          : recompile_choices
                                       := as_necessary );
```

Description

output : pathname_type
Specifies the name of the generated command file.

body_ind : yes_no_answer := no
specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the body_ind parameter need not be specified.

bodies_only : yes_no_answer := no
Controls whether all new units of the closure are compiled (default) or only the secondary units. This parameter is only effective if conditional => no is specified.

check : yes_no_same_answer := same
check => same means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the compile_host command. Otherwise the given value for the check parameter is included in the command file. By default the parameter value of the last compilation is included.

closure : closure_choices := execute
Controls the kind of the closure which is built and which is the basis for the investigation for new sources. closure => noclosure means that only the specified units are checked. closure => compile means that only those units on which the specified unit(s) transitively depend(s) are regarded. closure => execute means that - in addition - all related secondary units and the units they depend on are regarded. If closure => tree is specified, a warning is issued stating that this is not meaningful for this command and that the default value is taken instead.
By default, the execution closure is investigated for new sources.

conditional : yes_no_answer := yes
Controls whether the check for new sources is performed (default). no means that all units in the closure are compiled disregarding the modification date. This parameter is useful for compiling the complete closure with different parameters than the last time.

copy_source : yes_no_answer := yes
This parameter is included in the generated command file and thus affects the generated compile_host command. See the same parameter with the compile_host command. This parameter has no effect for the recompilation of obsolete units in accordance with the recompile_host command where copy_source => yes cannot be specified.

inline : yes_no_same_answer := same

command after the run of the command file generated by the autocompile_host command.

End of Command Description

4.3.3 First compilation

The SYSTEAM Ada System supports the first compilation of sources for which no compilation order is known by the compile_host command with parameter analyze_dependency in combination with the autocompile_host command.

With the analyze_dependency parameter the Compiler accepts sources in any order and performs the syntax analysis. If the sources are syntactically correct the units which are defined by the sources are entered into the library. Their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the autocompile_host command, i.e. the Autocompiler computes the first compilation order for the new sources. The name of the main program, of course, must be known and specified with the autocompile_host command.

Note that the compile_host (..., analyze_dependency => yes, ...) command replaces other units in the library with the same name as a new one. Thus the library may be modified even if the new units contain semantic errors; but the errors will not be detected until the command file generated by the autocompile_host command is run. Hence it is recommended to use an empty sublibrary if you do not know anything about the set of new sources.

If there are several sources containing units with the same name the last analyzed one will be kept in the library.

The autocompile_host command issues special warnings if the information about the new units is incomplete or inconsistent.

All error messages are self-explanatory. If a source line contains errors, the error messages for that source line are printed immediately below it. The exact position in the source to which an error message refers is marked by a number. This number is also used to relate different error messages given for one line to their respective source positions.

In order to enable semantic analysis to be carried out even if a program is syntactically incorrect, the Compiler corrects syntax errors automatically by inserting or deleting symbols. The source positions of insertions/deletions are marked with a vertical bar and a number. The number has the same meaning as above. If a larger region of the source text is affected by a syntax correction, this region is located for the user by repeating the number and the vertical bar at the end as well, with dots in between these bracketing markings.

A complete Compiler listing follows which shows the most common kinds of error messages, the technique for marking affected regions and the numbering scheme for relating error messages to source positions. It is slightly modified so that it fits into the page width of this document:

```
**********************************************************************
**                                                                  **
** SYSTEAM ADA - COMPILER       SUN3/SUNOS/CAIS x SUN3/SUNOS  1.82  **
**                                                                  **
** 90-01-29/08:39:44                                                **
**                                                                  **
**********************************************************************

======================================================================
=                                                                    =
=                                     Started at  : 08:39:44         =
=                                                                    =
=                                                                    =
=  PROCEDURE  LISTING_EXAMPLE                                        =
=                                                                    =
    1        PROCEDURE listing_example IS
    2        abc : procedure integer RANGE 0 .. 9 := 10E-1;
                   |1......1|
                                                      1
>>>>>  SYNTAX ERROR
          Symbol(s)  deleted (1)
>>>>>  SYMBOL ERROR (1)     An exponent for an integer literal must not
                            have a minus sign
    3        def integer RANGE 0 .. 9;
                |1
>>>>>  SYNTAX ERROR
          Symbol(s) inserted (1):  :
```

5 Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The SunOS system linker is used by the SYSTEAM Ada Linker.

To link a program, call the link_host command.

link_host Command Description

Format

```
PROCEDURE link_host (

    unit              : unitname_type     ;
    executable        : pathname_type     ;
    --not used:
    external          : string            := "";
    check             : yes_no_answer     := yes;
    complete          : yes_no_answer     := yes;
    debug             : yes_no_answer     := yes;
    inline            : yes_no_answer     := yes;
    library           : pathname_type
                                   := default_library;
    linker_options    : string            := "";
    linker_listing    : pathname_type     := nolist;
    list              : pathname_type     := nolist;
    log               : pathname_type     := nolog;
    machine_code      : yes_no_answer     := no;
    --not used:
    map               : pathname_type     := nomap;
    optimize          : yes_no_answer     := yes;
    relocatable       : yes_no_answer     := no;
    selective         : yes_no_answer     := no);
```

Description

The link_host command invokes the SYSTEAM Ada Linker.

The Linker builds an executable image which can be started by exporting the file contents to a SunOS file and executing that file with SunOS means

This parameter can be used to supply additional options for the call of `ld`. See the specification of the `ld` call following this command description.

`linker_listing : pathname_type := nolist`
Unless `linker_listing => nolist` is specified, the Linker of the SYSTEAM Ada System produces a listing in the given file containing a table of symbols which are used for linking the Ada units. This table is helpful when debugging an Ada program with the SunOS debugger.
By default, the Linker does not produce a listing.

`list : pathname_type := nolist`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`log : pathname_type := nolog`
This parameter controls whether the command (including the command used to call the system linker *ld*(1)) writes additional messages onto the specified file, and is also passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`machine_code : yes_no_answer := no`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command. If `linker_listing` is unequal `nolist` and `machine_code => yes` is specified, the Linker of the SYSTEAM Ada System appends a listing with the machine code of the program starter to the file given by `linker_listing`. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.
By default, no machine code listing is generated.

`map : pathname_type := nomap`
This parameter is not considered by this SYSTEAM Ada System.

`optimize : yes_no_answer := yes`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`relocatable : yes_no_answer := no`
`relocatable => yes` suppresses the generation of an executable object file. In this case the generated object file contains the code of all compilation units written in Ada and of those object modules of the predefined language environment and of the Ada run time system which are used by the main program; references into the Standard C library remain unresolved. The generated object module is suitable for further *ld*(1) processing. The name of its entry point is `_main`.

`selective : yes_no_answer := no`

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Enviroment Description (chapter 13 page 95 ff of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

```
PACKAGE standard IS

  TYPE boolean IS (false, true);

  -- The predefined relational operators for this type are as follows:

  -- FUNCTION "="   (left, right : boolean) RETURN boolean;
  -- FUNCTION "/="  (left, right : boolean) RETURN boolean;
  -- FUNCTION "<"   (left, right : boolean) RETURN boolean;
  -- FUNCTION "<="  (left, right : boolean) RETURN boolean;
  -- FUNCTION ">"   (left, right : boolean) RETURN boolean;
  -- FUNCTION ">="  (left, right : boolean) RETURN boolean;

  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:

  -- FUNCTION "AND" (left, right : boolean) RETURN boolean;
  -- FUNCTION "OR"  (left, right : boolean) RETURN boolean;
  -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;

  -- FUNCTION "NOT" (right : boolean) RETURN boolean;

  -- The universal type universal_integer is predefined.

  TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : integer) RETURN boolean;
  -- FUNCTION "/=" (left, right : integer) RETURN boolean;
  -- FUNCTION "<"  (left, right : integer) RETURN boolean;
```

```
-- FUNCTION "+"   (left, right : float) RETURN float;
-- FUNCTION "-"   (left, right : float) RETURN float;
-- FUNCTION "*"   (left, right : float) RETURN float;
-- FUNCTION "/"   (left, right : float) RETURN float;

-- FUNCTION "**"  (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type  universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE short_float IS DIGITS 6 RANGE
        - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

TYPE long_float IS DIGITS 18 RANGE
        - 16#0.FFFF_FFFF_FFFF_FFFF#E4096 ..
          16#0.FFFF_FFFF_FFFF_FFFF#E4096;

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character  literals corresponding to control characters are not
-- identifiers.

TYPE character IS
     (nul, soh, stx, etx,    eot, enq, ack, bel,
      bs,  ht,  lf,  vt,     ff,  cr,  so,  si,
```

```
    percent    : CONSTANT character := '%';
    ampersand  : CONSTANT character := '&';
    colon      : CONSTANT character := ':';
    semicolon  : CONSTANT character := ';';
    query      : CONSTANT character := '?';
    at_sign    : CONSTANT character := '@';
    l_bracket  : CONSTANT character := '[';
    back_slash : CONSTANT character := '\';
    r_bracket  : CONSTANT character := ']';
    circumflex : CONSTANT character := '^';
    underline  : CONSTANT character := '_';
    grave      : CONSTANT character := '`';
    l_brace    : CONSTANT character := '{';
    bar        : CONSTANT character := '|';
    r_brace    : CONSTANT character := '}';
    tilde      : CONSTANT character := '~';

    lc_a : CONSTANT character := 'a';
    ...
    lc_z : CONSTANT character := 'z';

  END ascii;

  -- Predefined subtypes:

  SUBTYPE natural  IS integer RANGE 0 .. integer'last;
  SUBTYPE positive IS integer RANGE 1 .. integer'last;

  -- Predefined string type:

  TYPE string IS ARRAY(positive RANGE <>) OF character;

  PRAGMA pack(string);

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : string) RETURN boolean;
  -- FUNCTION "/=" (left, right : string) RETURN boolean;
  -- FUNCTION "<"  (left, right : string) RETURN boolean;
  -- FUNCTION "<=" (left, right : string) RETURN boolean;
  -- FUNCTION ">"  (left, right : string) RETURN boolean;
  -- FUNCTION ">=" (left, right : string) RETURN boolean;

  -- FUNCTION "&" (left : string;    right : string)    RETURN string;
  -- FUNCTION "&" (left : character; right : string)    RETURN string;
  -- FUNCTION "&" (left : string;    right : character) RETURN string;
  -- FUNCTION "&" (left : character; right : character) RETURN string;
```

13.3.1 The Package COLLECTION_MANAGER

In addition to unchecked storage deallocation (cf. LRM(§13.10.1)), this implementation provides the generic package collection_manager, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

```
GENERIC
   TYPE elem IS LIMITED PRIVATE;
   TYPE acc  IS ACCESS elem;
PACKAGE collection_manager IS

   TYPE status IS LIMITED PRIVATE;

   PROCEDURE mark (s : OUT status);

      -- Marks the heap of type ACC and
      -- delivers the actual status of this heap.

   PROCEDURE release (s : IN status);

      -- Restore the status s on the collection of ACC.
      -- RELEASE without previous MARK raises CONSTRAINT_ERROR

   PROCEDURE reset;

      -- Deallocate all objects on the heap of ACC

   PRIVATE
      -- private declarations

END collection_manager;
```

A call of the procedure release with an actual parameter s causes the storage occupied by those objects of type acc which were allocated after the call of mark that delivered s as result, to be reclaimed. A call of reset causes the storage occupied by all objects of type acc which have been allocated so far to be reclaimed and cancels the effect of all previous calls of mark.

```
PACKAGE command_arguments IS

   argc : integer;
      -- number of arguments of a command

   FUNCTION argl (nr : natural) RETURN natural;
      -- length of nr-th argument

   FUNCTION argv (nr : natural) RETURN string;
      -- value of nr-th argument

   envc : integer;
      -- number of values in environment vector

   FUNCTION envl (nr : natural) RETURN natural;
      -- length of nr-th value in environment vector

   FUNCTION envv (nr : natural) RETURN string;
      -- value of nr-th value in environment vector

END command_arguments;
```

A call of the function **argv** with actual parameter **nr** delivers the **nr**-th argument $(0 <= nr <= argc - 1)$ of the command which executed the Ada program. **argc** returns the number of arguments passed to the command. **argv (0)** returns the name of the command which is currently being executed. Each of the arguments accessed through **argv (nr)** is represented as a string of length **argl (nr)**.

Information about the environment of the process that executes the command may be obtained in an analogous manner. The function **envv (nr)** $(0 <= nr <= envc - 1)$ delivers a string of the form "name=value" as specified in *environ*(5). The string length may be obtained by calling **envl (nr)**.

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
    has no effect.

ELABORATE
    is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit u was given, if the compiled unit contains an instantiation of u (or of a generic program unit in u) and if it is clear that u *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
    Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

0 .. 15, as declared in the predefined library package `system` (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA `priority` for it is the same as if the PRAGMA `priority` 0 had been given (i.e. the task has the lowest priority).

SHARED
is fully supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §15.1.2 for the implementation-defined PRAGMA `suppress_all`.

SYSTEM_NAME
has no effect.

15.1.2 Implementation-Defined Pragmas

BYTE_PACK
see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)
<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the `call` instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.
Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. The user should not define external names beginning with an underline because Compiler generated names as well as external symbols of C (for example SunOS/CAIS system calls) are

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are the same as those used for subprograms for which a `PRAGMA interface (ASSEMBLER,...)` is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada; it is provided in object form using the `-ld` option with the `sas.link` (or `sas.c` or `sas.make`) command.

> In many cases it is more convenient to follow the C procedure calling standard. Therefore the SYSTEAM Ada System provides the `PRAGMA interface(c,...)`, which supports the standard return of the function result and the standard register saving. This pragma is described in the next section.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.
For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the **JSR** instruction. The address pointing to the beginning of the parameter block is pushed onto the stack before calling the subprogram.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the

Ordering of parameters:

The ordering of the parameters in the parameter block is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address `a6@(8)`.

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs **pack** or **byte_pack** (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by

```
link    a6,-(#<frame-size>+4)
clrl    a6@(-4)
        | The field at address a6@(-4) is reserved
        | for use by the Ada runtime system
```

The return code sequence is then simply

```
rts
```

for procedures without parameters and

```
rtd     #4
```

for functions and procedures with parameters.

15.1.4 Pragma Interface(C,...)

The SYSTEAM Ada System supports `PRAGMA interface(C,...)`.

With the help of this pragma *and* by obeying some rules (described below) subprograms can be called which follow the C procedure calling standard. As the user must know something about the internal calling conventions of the SYSTEAM Ada System we recommend reading §15.1.3 before reading this section and before using `PRAGMA interface(C,...)`.

For each Ada subprogram for which

```
PRAGMA interface (C, <ada_name>);
```

is specified, a routine implementing the body of the subprogram <ada_name> must be provided, written in any language that obeys the C calling conventions (cf. SunOS Documentation Set, C Programmer's Guide, Chapter D.3), in particular:

- Saving registers
- Calling mechanism
- C stack frame format.

SunOS/CAIS system calls or subroutines are allowed too.

```
                              oflag : integer) RETURN integer;
      PRAGMA interface (C, unix_open);
      PRAGMA external_name ("_open", unix_open);

   BEGIN
      ret_code := unix_open (file_name'address, read_mode);
      IF ret_code = -1 THEN
         RAISE use_error;
      END IF;
   END unix_call;
```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.
If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.
If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.
For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding uppercase character. For example, `character'image(`*nul*`) = "NUL"`.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;

   TYPE address IS ACCESS designated_by_address;
   FOR address'storage_size USE 0;

   address_zero : CONSTANT address := NULL;

   TYPE name IS (sun3_sunos);

   system_name  : CONSTANT name := sun3_sunos;

   storage_unit : CONSTANT := 8;
   memory_size  : CONSTANT := 2 ** 31;
   min_int      : CONSTANT := - 2 ** 31;
   max_int      : CONSTANT := 2 ** 31 - 1;
   max_digits   : CONSTANT := 18;
   max_mantissa : CONSTANT := 31;
   fine_delta   : CONSTANT := 2.0 ** (-31);
   tick         : CONSTANT := 0.02;

   SUBTYPE priority IS integer RANGE 0 .. 15;

   FUNCTION "+" (left : address; right : integer) RETURN address;

   FUNCTION "+" (left : integer; right : address) RETURN address;

   FUNCTION "-" (left : address; right : integer) RETURN address;

   FUNCTION "-" (left : address; right : address) RETURN integer;

   SUBTYPE external_address IS STRING;

   -- External addresses use hexadecimal notation with characters
   -- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
   --    "7FFFFFFF"
   --    "80000000"
   --    "8" represents the same address as "00000008"

   FUNCTION convert_address (addr : external_address) RETURN address;

      -- CONSTRAINT_ERROR is raised if the external address ADDR
      -- is the empty string, contains characters other than
      -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
      -- value cannot be represented with 32 bits.
```

```
-- correspond to any situation covered by Ada, e.g.:
--    illegal instruction encountered
--    error during address translation
--    illegal address

TYPE exception_id IS NEW address;

no_exception_id     : CONSTANT exception_id := address_zero;

-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id    : CONSTANT exception_id := ... ;
program_error_id    : CONSTANT exception_id := ... ;
storage_error_id    : CONSTANT exception_id := ... ;
tasking_error_id    : CONSTANT exception_id := ... ;

non_ada_error_id    : CONSTANT exception_id := ... ;

status_error_id     : CONSTANT exception_id := ... ;
mode_error_id       : CONSTANT exception_id := ... ;
name_error_id       : CONSTANT exception_id := ... ;
use_error_id        : CONSTANT exception_id := ... ;
device_error_id     : CONSTANT exception_id := ... ;
end_error_id        : CONSTANT exception_id := ... ;
data_error_id       : CONSTANT exception_id := ... ;
layout_error_id     : CONSTANT exception_id := ... ;

time_error_id       : CONSTANT exception_id := ... ;

TYPE exception_information IS
   RECORD
      excp_id          : exception_id;

         -- Identification of the exception. The codings of
         -- the predefined exceptions are given above.

      code_addr        : address;

         -- Code address where the exception occurred. Depending
         -- on the kind of the exception it may be be address of
         -- the instruction which caused the exception, or it
         -- may be the address of the instruction which would
         -- have been executed if the exception had not occurred.

      error_code       : integer;
```

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation` (`release`) on a collection, all following calls of `release` (`unchecked deallocation`) until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.